Chapter 10

Conclusions

The m3-- language enhances software reusability by the introduction of polymorphic types. Although polymorphism as such is available in other languages, little effort seems to have been spent in making that polymorphism explicit. This led to an unfortunate amount of exceptional rules and ambiguities. The rules were introduced to keep the use of polymorphic types within the limits of easy implementation. The ambiguities are a direct effect of the implicitness of the polymorphism.

Exceptional rules are unfortunate, since they cloud a language's description. The applicability of certain language elements becomes context-dependent, increasing the amount of work needed to fully comprehend a programming language's description. These exceptional rules may also lead to programs in which unexpected approaches have been taken to avoid such exceptional situations.

10.1 Polymorphism in Programming

Polymorphism in programming languages is largely implemented by ad hoc language structures. In some cases special syntactic rules have been created to allow polymorphic code to be constructed in only a few situations. A typical example of this is Pascal's conformant array concept. Other languages have generalized this idea into open arrays, but their use is restricted to prevent implementation problems.

The typed λ-calculus has very good models for dealing with polymorphism, but somehow these models have not yet found their way into languages in practical use. Recent work by Abadi and Cardelli[3] shows that the problems of object types are still under discussion, and not all implications of such types as `Current`, as it is incorporated in m3-- (and also available in Eiffel), are fully understood.

Another line of research which shows promising connections with such type systems as that of m3-- is the work on 'formulas as types', where predicate logic may be used to prove or disprove certain relations between types. It might be very useful if mathematical proofs for certain relations between types could be constructed.

It is interesting to note that many programming language designers have embraced the idea of type tags. The performance cost is not very great, and in an object oriented environment they are very useful to check the actual type of an object. In these languages however, the presence of type tags –or -codes– is presented as a safety enhancing *feature*, and their availability seems more an environmental influence than an actual aspect of the type system. In Modula-3 the availability of type codes is combined with the availability of a garbage collector, and for those wanting to write an application which does not 'need' garbage collection and type codes, a special 'untraced' heap

is provided. The need for type codes however can be described perfectly with the polymorphic types of m3--, and as such can be explained in the light of the type system.

Although programmers tend to prefer languages with a compact notation, the realization that this same compactness often makes code cryptic –and hence difficult to maintain– is growing. The LILLEANNA[51] project shows the clear turn to more formal and well-specified ways of constructing programs. Philips, a major electronics company in the Netherlands, is known to have sent its researchers to production departments, hoping to improve software quality and production by spreading the word on formal specification languages.

10.2 The m3-- Language

M3-- as a language will probably need a number of additional language features before it would be useful as an actual programming language. Not only such things as floating-point numbers, sets and packed types, but also exceptions, which have proven their value in enhancing a language's ability to deal with unexpected or unwanted states. Having a language with many (sometimes long) keywords is not as much of a problem as one might expect, even though it is a constant point of criticism by (especially) C and C++ programmers with regard to languages such as Modula-3.

Using (sometimes complicated) type expressions may seem rather unattractive at first. Also, although explicit polymorphism will allow an improvement in code quality, it is not likely to reduce the number of places where polymorphism appears. Rather, it makes polymorphism visible and allows the 'flow of types' to be used. Instead of saying that "a `List` is taken, and a `List` is returned," it is possible to say "any extension of `List` is taken, and a value of that same extension is returned." It is here that the profits are harvested, both in a reduction of necessary type tests, as well as in an enhanced quality because of a more precise specification.

Replacing source-level polymorphism with runtime polymorphism may be more expensive at times, but it already exists in many places (cf. open arrays, 'generic pointers') and by taking care of the basic concept, many more valuable uses become possible. It should be understood however that the programmer has a certain responsibility in choosing between efficiency and reusability.

10.3 The Implementation of m3--

The implementation of m3-- has –unfortunately– not progressed far enough to be able to actually produce code. The compiler as it now exists is however able to do type inference for generic procedures and opaque parameters. It also manages generic types and generic instantiation well. It can take an interface or module, and give judgements on the need for type checks.

10.4 Reusability of Code

Since this thesis started with software reuse, I will now close the circle by ending with it. The major soul-searching question is whether or not the use of m3-- will actually allow an enhancement of the reusability of code. Unfortunately, software construction being the craftsmanship it is, this is mainly dependent on the use a programmer would make of it. The old aphorism "A real programmer can write assembler in any language" shows the basic problem: Software reuse and -reusability still is mostly a matter of what the programmer can do, rather than in what language

he will do it. As was concluded in chapter 1, many of the problems in software reuse are human problems. Providing a methodology and formalism will only help if they are adhered to, and even then the largest impact will be provided by the individual's capabilities.

What *can* be asked is whether or not `m3--` provides the programmer with an adequate tool for writing highly reusable code, and a positive answer to that question is what I hope to have conveyed to the reader. The key issues are parameterization and genericism, and `m3--` has a type system in which these are well represented. I leave it to the reader to draw his own conclusions, and will welcome any comments and discussion.

10.5 Further Research

The obvious continuation for this project would be the completion of both `m3--` as a 'real' programming language, as well as the completion of the compiler. Another line of research that merits attention is the work on 'formulas as types', and its implications for 'real' type systems.